Universidad Abierta Interamericana

Campus Virtual UAIOnline

LENGUAJES DE ÚLTIMA GENERACIÓN

ESTRATEGIA DE ALGORITMOS

UNIDAD 5/CLASE ACTUAL:
ESTRATEGIA DE ALGORITMOS/CLASE 14

Autor de contenidos:
Mauricio Prinzo

PRESENTACIÓN

A través de los años, los científicos de la computación han identificado diversas técnicas generales que a menudo producen algoritmos eficientes para la resolución de muchas clases de problemas.

En esta unidad presentaremos algunas de las técnicas más importantes.

1. INTRODUCCIÓN – ESTRATEGIA DE DISEÑO

El desarrollo de algoritmos es un tema fundamental en el diseño de programas o soluciones. Por lo cual, el alumno debe tener buenas bases que le sirvan para poder crear de manera fácil y rápida sus programas.

La computadora es una máquina que por sí sola no puede hacer nada, necesita ser programada, es decir, introducirle instrucciones u órdenes que le digan lo que tiene que hacer.

Un programa es la solución a un problema inicial, así que todo comienza ahí: en el Problema.

El problema se analiza teniendo presente la especificación de los requisitos dados por el cliente de la empresa o por otra persona que encarga el programa.

Una vez analizado el problema, se diseña una solución que conducirá a un ***algoritmo*** que resuelva el problema.

Se puede definir Algoritmo como: "Método para resolver un problema mediante una serie de pasos **precisos, definidos y finitos**"

1.1. RECURSIVIDAD

La recursividad es una técnica fundamental en el diseño de algoritmos eficientes, que está basada en la solución de versiones más pequeñas del problema, para obtener la solución general del mismo. Una instancia del problema se soluciona según la solución de una o más instancias diferentes y más pequeñas que ella.

Es una herramienta poderosa que sirve para resolver cierto tipo de problemas reduciendo la complejidad y ocultando los detalles del problema. Esta herramienta consiste en que una función o procedimiento se llama a sí mismo.

Una gran cantidad de algoritmos pueden ser descritos con mayor claridad en términos de recursividad, típicamente el resultado será que sus programas serán más pequeños.

La recursividad es una alternativa a la iteración o repetición, y aunque en tiempo de computadora y en ocupación en memoria es la solución recursiva menos eficiente que la solución iterativa, existen numerosas situaciones en las que la recursividad es una solución simple y natural a un problema que en caso contrario ser difícil de resolver. Por esta razón se puede decir que la recursividad es una herramienta potente y útil en la resolución de problemas que tengan naturaleza recursiva y, por ende, en la programación.

La característica importante de la recursividad es que siempre existe un medio de salir de la definición, mediante la cual se termina el proceso recursivo.

Ventajas:

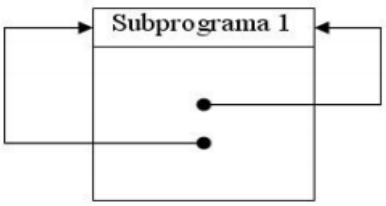
- Puede resolver problemas complejos.
- Solución más natural.

Desventajas:

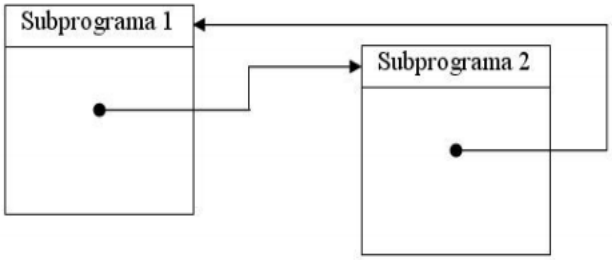
- Se puede llegar a un ciclo infinito.
- Versión no recursiva más difícil de desarrollar.
- Para la gente sin experiencia es difícil de programar.

Tipos de Recursividad

Directa o simple: un subprograma se llama a si mismo una o más veces directamente.



Indirecta o mutua: un subprograma A llama a otro subprograma B y éste a su vez llama al subprograma A.



1.2 DIVIDE PARA CONQUISTAR

Dividir para Conquistar Muchos algoritmos útiles tienen una estructura recursiva, de modo que para resolver un problema se llaman recursivamente a sí mismos una o más veces para solucionar subproblemas muy similares.

Esta estructura obedece a una estrategia dividir-y-conquistar, en que se ejecuta tres pasos en cada nivel de la recursión:

• **Dividir**. Dividen el problema en varios subproblemas similares al problema original pero de menor tamaño;

• **Conquistar.** Resuelven recursivamente los subproblemas si los tamaños de los subproblemas son suficientemente pequeños, entonces resuelven los subproblemas de manera directa; y luego,

• **Combinar.** Combinan estas soluciones para crear una solución al problema original.

La técnica **Dividir para Conquistar (o Divide y Vencerás)** consiste en descomponer el caso que hay que resolver en subcasos más pequeños, resolver independientemente los subcasos y por último combinar las soluciones de los subcasos para obtener la solución del caso original

1.3 Programación Dinámica

Frecuentemente para resolver un problema complejo se tiende a dividir este en subproblemas más pequeños, resolver estos últimos (recurriendo posiblemente a nuevas subdivisiones) y combinar las soluciones obtenidas para calcular la solución del problema inicial.

Puede ocurrir que la división natural del problema conduzca a un gran número de subejemplares idénticos. Si se resuelve cada uno de ellos sin tener en cuenta las posibles repeticiones, resulta un algoritmo ineficiente; en cambio sí se resuelve cada ejemplar distinto una sola vez y se conserva el resultado, el algoritmo obtenido es mucho mejor.

Esta es la idea de la programación dinámica: no calcular dos veces lo mismo y utilizar normalmente una tabla de resultados que se va rellenando a medida que se resuelven los subejemplares.

La programación dinámica es un método ascendente. Se resuelven primero los subejemplares más pequeños y por tanto más simples. Combinando las soluciones se obtienen las soluciones de ejemplares sucesivamente más grandes hasta llegar al ejemplar original.

1.4. Algoritmo Ávido

Los algoritmos ávidos o voraces (Greedy Algorithms) son algoritmos que toman decisiones de corto alcance, basadas en información inmediatamente disponible, sin importar consecuencias futuras.

Suelen ser bastante simples y se emplean sobre todo para resolver problemas de optimización, como por ejemplo, encontrar la secuencia óptima para procesar un conjunto de tareas por un computador, hallar el camino mínimo de un grafo, etc.

Habitualmente, los elementos que intervienen son:

• un conjunto o lista de **candidatos** (tareas a procesar, vértices del grafo, etc.);

• un conjunto de **decisiones** ya tomadas (candidatos ya escogidos);

• una función que determina si un conjunto de candidatos es una solución al problema (aunque no tiene por qué ser la óptima);

• una **función** que determina si un conjunto es completable, es decir, si añadiendo a este conjunto nuevos candidatos es posible alcanzar una **solución** al problema, suponiendo que esta exista;

• una **función** de selección que escoge el candidato aún no seleccionado que es más **prometedor**;

• una **función objetivo** que da el valor/coste de una solución (tiempo total del proceso, la longitud del camino, etc.) y que es la que se pretende maximizar o minimizar;

Para resolver el problema de optimización hay que encontrar un conjunto de candidatos que optimiza la función objetivo. Los algoritmos voraces proceden por pasos.

Inicialmente el conjunto de candidatos es vacío. A continuación, en cada paso, se intenta añadir al conjunto el mejor candidato de los aún no escogidos, utilizando la función de selección. Si el conjunto resultante no es completable, se rechaza el candidato y no se le vuelve a considerar en el futuro. En caso contrario, se incorpora al conjunto de candidatos escogidos y permanece siempre en él. Tras cada incorporación se comprueba si el conjunto resultante es una solución del problema. Un algoritmo voraz es correcto si la solución así encontrada es siempre óptima.

Ejemplo: Mínimo número de monedas

Se desea pagar una cantidad de dinero a un cliente empleando el menor número posible de monedas. Los elementos del esquema anterior se convierten en:

• **candidato:** conjunto finito de monedas de, por ejemplo, 1, 5, 10 y 25 unidades, con una moneda de cada tipo por lo menos;

• **solución:** conjunto de monedas cuya suma es la cantidad a pagar;

• **completable:** la suma de las monedas escogidas en un momento dado no supera la cantidad a pagar;

• **función de selección:** la moneda de mayor valor en el conjunto de candidatos aún no considerados;

• **función objetivo:** número de monedas utilizadas en la solución.

2. EFICIENCIAS

2.1 PRINCIPIO DE INVARIANCIA

Dado un algoritmo y dos implementaciones suyas I1 e I2, que tardan T1(n) y T2(n) segundos respectivamente, el Principio de Invarianza afirma que existe una constante real c > 0 y un número natural n0 tales que para todo n ≥ n0 se verifica que T1(n) ≤ cT2(n).

Es decir, el tiempo de ejecución de dos implementaciones distintas de un algoritmo dado no va a diferir más que en una constante multiplicativa.

Esto significa que el tiempo para resolver un problema mediante un algoritmo depende de la naturaleza del algoritmo y no de la implementación del algoritmo.

Cuando se quiere comparar la eficiencia temporal de dos algoritmos, tiene mayor influencia el tipo de función que la constante c.

2.2 TIEMPO DE EJECUCIÓN

El tiempo que requiere un algoritmo para dar una respuesta, se divide generalmente en 3 casos:

Peor Caso: caso más extremo, donde se considera el tiempo máximo para solucionar un problema.

Caso promedio: caso en el cual, bajo ciertas restricciones, se realiza un análisis del algoritmo

Mejor caso: caso ideal en el cual el algoritmo tomará el menor tiempo para dar una respuesta

Para medir T(n) usamos el número de operaciones elementales, las mismas pueden ser:

- Operación aritmética.
- Asignación a una variable.
- Llamada a una función.
- Retorno de una función.
- Comparaciones lógicas (con salto).
- Acceso a una estructura (arreglo, matriz, lista ligada…).

Se le llama tiempo de ejecución, no al tiempo físico, sino al número de operaciones elementales que se llevan a cabo en el algoritmo.

3. COMPLEJIDAD ALGORÍTMICA

3.1 MEDIDAS ASINTÓTICAS

Las cotas de complejidad, también llamadas **medidas asintóticas** sirven para clasificar funciones de tal forma que podamos compararlas. Las medidas asintóticas permiten analizar qué tan rápido crece el tiempo de ejecución de un algoritmo cuando crece el tamaño de los datos de entrada, sin importar el lenguaje en el que esté implementado ni el tipo de máquina en la que se ejecute.

Existen diversas notaciones asintóticas para medir la complejidad, las tres cotas de complejidad más comunes son: la notación O (o mayúscula), la notación Ω (omega mayúscula) y la notación θ (theta mayúscula) y todas se basan en el peor caso.

Cota superior asintótica: Notación O (o mayúscula): Dada una función f, se estudian todas aquellas funciones g que a lo sumo crecen tan deprisa como f. Al conjunto de tales funciones se le llama cota superior de f y lo denominamos O(f). Conociendo la cota superior de un algoritmo se puede asegurar que, en ningún caso, el tiempo empleado será de un orden superior al de la cota.

Cota inferior asintótica: Notación Ω (omega mayúscula): Dada una función f, se quieren estudiar aquellas funciones g que a lo sumo crecen tan lentamente como f. Al conjunto de tales funciones se le llama cota inferior de f y se denominan Ω(f). Conociendo la cota inferior de un algoritmo se puede asegurar que, en ningún caso, el tiempo empleado será de un orden inferior al de la cota.

Orden exacto o cota ajustada asintótica: Notación θ (theta mayúscula) Como última cota asintótica, están los conjuntos de funciones que crecen asintóticamente de la misma forma

Ejemplo: ¿Cuál es la complejidad de multiplicar dos enteros?

- Depende de cual sea la medida del tamaño de la entrada.
- Podrá considerarse que todos los enteros tienen tamaño O(1), pero eso no será útil para comparar este tipo de algoritmos.
- En este caso, conviene pensar que la medida es el logaritmo del número.
- Si por el contrario estuviésemos analizando algoritmos que ordenan arreglos de enteros, lo que importa no son los enteros en sí, sino cuántos tengamos.
- Entonces, para ese problema, la medida va a decir que todos los enteros miden lo mismo

3.2 PROGRAMACIÓN PARALELA Y SECUENCIAL

La programación secuencial es cuando **una tarea va después de otra**. Es un proceso lento en el que, si una tarea se retrasa, el sistema completo debe esperar. La ventaja es que es **fácil** de entender y de implementar.

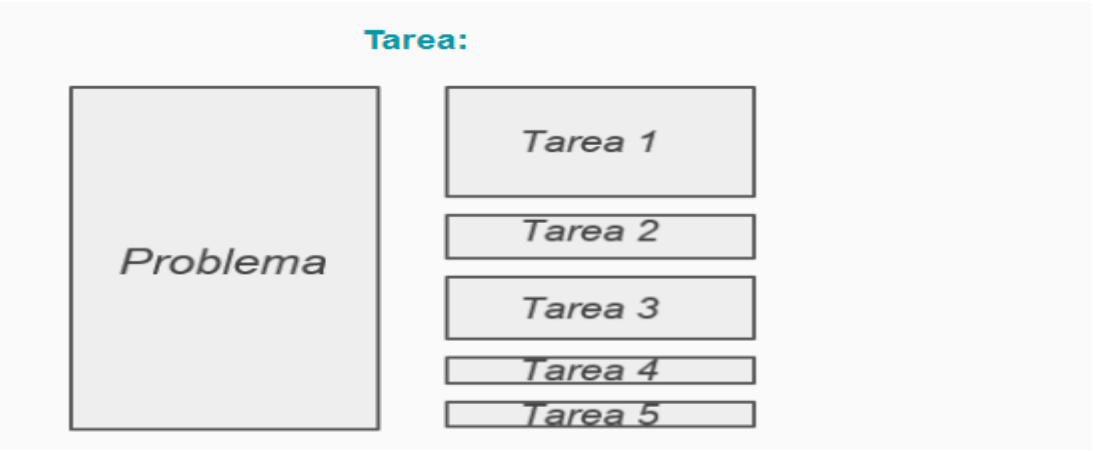
Mientras que **programación paralela** es **el uso de múltiples recursos** computacionales para resolver un problema.

En el sentido más simple, la programación paralela es el uso simultáneo de múltiples recursos computacionales para resolver un problema computacional:

- Un problema se divide en partes discretas que se pueden resolver simultáneamente
- Cada parte se descompone en una serie de instrucciones
- Las instrucciones de cada parte se ejecutan simultáneamente en diferentes procesadores
- Se emplea un mecanismo global de control/coordinación

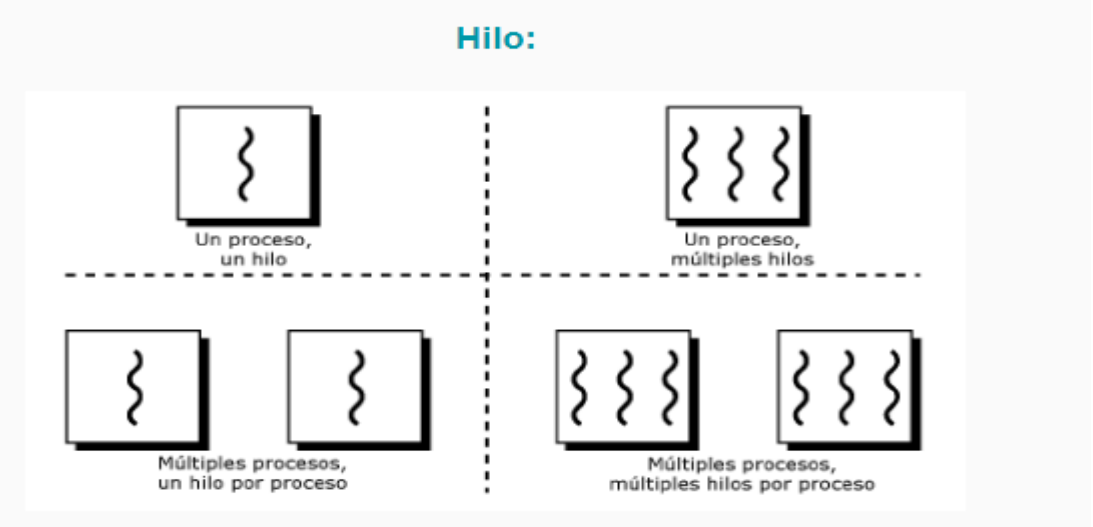
Conceptos acerca de Tareas

Un problema complejo se subdivide en una **cantidad discreta** de tareas que representan trabajo computacional. Una tarea está compuesta de un **conjunto de instrucciones** que serán ejecutadas por un procesador.



Conceptos acerca de Hilos

Un proceso pesado padre puede convertirse en varios **procesos livianos hijos**, ejecutados de manera concurrente. Cada uno de estos procesos livianos se conoce como hilo. Estos se comunican entre ellos a través de la memoria global.



OpenMP, es una interfaz de programa de aplicación (API) que se puede utilizar para dirigir explícitamente paralelismo de memoria compartida multi-procesos. Está compuesto por:

- Directivas de compilación
- Runtime Library Routines
- Variables de entorno

```
#include <omp.h>

int main () {

int var1, var2, var3;

//Código en serie...

//Comienzo de la región paralela hace fork del conjunto de threads

#pragma omp parallel private(var1, var2) shared(var3)

{       //Región paralela ejecutada por todos los threads
        //otras directivas OpenMP
        //Todos los threads se juntan en el thread master
}

//Continuación del código...

}
```

3.3 Algoritmo Distribuido

Un algoritmo distribuido es un tipo específico de algoritmo que se utiliza en sistemas distribuidos y que ha sido diseñado para aprovechar las características de este tipo de sistemas.

Propiedades de algoritmos distribuidos:

- La información relevante se distribuye entre varias máquinas.
- Se toman decisiones sólo en base a la información local.
- Debe evitarse un punto único de fallo.
- No existe un reloj común.

Problemas a considerar:

- Tiempo y estados globales: sincronización en tiempo real.
- Exclusión mutua: Mecanismo de coordinación entre varios procesos concurrentes a la hora de acceder a recursos/secciones compartidas
- Algoritmos de elección
- Operaciones atómicas distribuidas: Transacciones

3.4 Fases de resolución de problemas

La **resolución de un problema** consiste en el proceso que a partir de la descripción de un problema, expresado habitualmente en lenguaje natural y en términos propios del dominio del problema, permite desarrollar un programa que resuelva dicho problema.

Este proceso exige los siguientes pasos:

1. **Análisis del problema**: El problema se analiza teniendo presente la especificación de los requisitos dados por el cliente de la empresa o por otra persona que encarga el programa.
2. **Diseño o desarrollo de un algoritmo:** una vez analizado el problema, se diseña una solución que conducirá a un *algoritmo* que resuelva el problema
3. **Codificación (implementación)** Esta etapa consiste en transcribir o adaptar el algoritmo a un lenguaje de programación, se tendrá que adaptar todos los pasos diseñados en el algoritmo con sentencias y sintaxis propias del lenguaje.
4. **Ejecución, verificación y depuración:** el programa se ejecuta, se comprueba rigurosamente y se elimina todos los errores (denominados "*bugs"*, en inglés) que puedan aparecer
5. **Mantenimiento:** El programa se actualiza y modifica, cada vez que sea necesario, de modo que se cumplan todas las necesidades de cambio de sus usuarios.

4. CONCLUSIÓN

En esta clase hemos visto las diferentes técnicas utilizada por aloritmos. para trabajar graficar con facilidad. En todo sistema informatizado, el uso de estos servicios nutre de riqueza y potencialidad a los desarrollos al facilitar el uso de los medios visuales que colaboran en la toma de decisiones en el área gerencial de una empresa.

Para ampliar los contenidos, le sugerimos la lectura de la bibliografía

- Brassard, G.; Bratley, P.. Fundamentos de algoritmia.-- Madrid : Prentice Hall, 1997. xiii, 579 páginas

- Rosa Guerequeta, Antonio Vallecillo. (1998). Técnicas de Diseño de Algoritmos. Universidad de malaga.

- Maria, Gopmez fuetes. Jorge, Cervantes. (2014). Introducción al análisis y diseño de algoritmos. Universidad autónoma metropolitana.